

REDUCING PROGRAMMING COMPLEXITY IN APPLICATIONS INTERFACING WITH PARSERS FOR DATA ELEMENTS REPRESENTED ACCORDING TO A MARKUP LANGUAGE

DESCRIPTION

(Para 1) Background of the Invention

(Para 2) Field of the Invention

(Para 3) The present invention relates to applications programming environment using markup languages such as XML, and more specifically to reducing programming complexity in applications interfacing with parsers for data elements represented according to a markup languages.

(Para 4) Related Art

(Para 5) Markup languages such as XML are generally used to represent various data of interest. A typical markup language generally contains tags which indicate one or more of various aspects such as what the data represents or how the data is to be displayed, etc. For example, in XML, relevant data is enclosed between a start tag and an end tag, indicating what the enclosed data represents. The relevant data and the tags are referred to as data elements in the present application.

(Para 6) Applications, which require data represented according to markup languages, often interface with a parser for the various data elements of interest. In a typical scenario, the XML data is stored in an XML data file, and the parser retrieves the data elements and provides the retrieved elements to the applications according to a pre-specified approach.

(Para 7) According to one pre-specified approach often referred to as 'pull parsing', an application generally requests that the 'next' data element be provided. In response, a parser retrieves (e.g., from an XML document) the next data element (in sequential order) from the XML data file and provides the next data element to the application. Since the data elements are generally retrieved in sequential order, the parsers are referred to as sequential parsers.

(Para 8) According to another sequential parsing approach, often referred to as 'push parsing', a parser retrieves data elements in an XML data file without necessarily receiving a request for a next data element, and "pushes" the retrieved data element to the application (after the application has specified the file identifier of the XML data file). The applications are designed to process such data elements received from the push-based parsers. SAX and XNI are the two industry standards, which support push parsing.

(Para 9) The pull and push based parses are broadly referred to as event based parsers since the requests (from application) of pull parsing and the pushing of data elements in push parsing can be viewed as events. It may be appreciated that the data elements are provided one at a time in event based parsing techniques.

(Para 10) In another broad prior approach, commonly referred to as ' Object based parsing ', the parser generally creates a hierarchical representation of data elements in an XML data file while parsing the XML data file and saves the hierarchical representation (in the form of a data structure) of the data elements into a random access memory (RAM) which is accessible to the application. The memory resident data structure is referred to as DOM (Document Object Model). The object based parsers return the DOM to the application, typically after parsing of the XML data file is complete. Thus, the applications are designed to access the RAM for any desired data element thereafter. Two commonly used DOM standards are W3C DOM and J-DOM.

(Para 11) An advantage of the object based parsing over the event parsers is that the data elements are available quickly to the applications. However, the memory (RAM) requirements are substantially more since a data structure representing the entire XML data file may be saved in the RAM.

(Para 12) Applications often require an identifier ("portion identifier") of portions (containing one or more data elements) of a data file. In the case of XML, the portion identifier is referred to as an XPath, and is defined in a hierarchical fashion similar to the file paths in various computer systems. The portion identifier may be required, for example, to determine a parent/ancestor of a data element. As an illustration, assuming that an XML data file contains data related to a school and that a retrieved data element corresponds to the name of a student of a section, and it is desirable to determine the teacher of the section. The name of the teacher may be structured as an ancestor of the retrieved data element, and accordingly it may be desirable for an application to have the XPath of the name of the student.

(Para 13) In a prior approach, applications may include program logic to build/construct XPath (or portion identifier, in general) of such desired parts of an XML data file. The need to build such portion identifiers of data elements of interest generally adds to the programming complexity of applications. At least for such a reason, there is a general need to reduce programming complexity in applications interfacing with parsers for data elements represented according to markup languages.

(Para 14) Brief Description of the Drawings

(Para 15) The present invention will be described with reference to the accompanying drawings briefly described below.

(Para 16) Figure (Fig.) 1 is a block diagram illustrating the details of a general approach using which various aspects of the present invention are implemented in one embodiment.

(Para 17) Figure 2 is a flow chart illustrating the manner in which a parser of a data file (containing data according to a markup language) can be implemented to simplify the implementation of applications according to an aspect of the present invention.

(Para 18) Figure 3 is a flow chart illustrating the manner in which the implementation of applications is simplified according to an aspect of the present invention.

(Para 19) Figure 4 contains XML data used to illustrate the operation of example embodiments implemented according to various features of the present invention.

(Para 20) Figure 5A contains Java code providing an outline for the implementation of an application interfacing with an object-based parser, which provides portion identifiers according to various aspects of the present invention.

(Para 21) Figure 5B contains Java code illustrating the manner in which the output data structure generated by an object-based parser can be accessed by an application.

(Para 22) Figure 6 contains Java Code for an application interfacing with a event-based push parser (e.g., SAX parser) according to an aspect of the present invention.

(Para 23) Figure 7 contains Java Code for an application interfacing with an event-based pull parser according to an aspect of the present invention.

(Para 24) Figures 8A-8F are shown containing tables illustrating a custom application programming interface (API) provided by an example parser in an alternative embodiment of the present invention.

(Para 25) Figure 9 illustrates the manner in which the APIs of Figures 8A-8F can be used to implement applications according to various aspects of the present invention.

(Para 26) Figures 10A, 10B, 10C and 10D illustrate the XPath's and values returned by parsers providing the API of Figures 8A-8F in an embodiment.

(Para 27) Figure 11 illustrates the manner in which the present invention may be implemented in the form of a software.

(Para 28) In the drawings, like reference numbers generally indicate identical, functionally similar, and/or structurally similar elements. The drawing in which an element first appears is indicated by the leftmost digit(s) in the corresponding reference number.

(Para 29) Detailed Description

(Para 30) 1. Overview

(Para 31) In an embodiment of the present invention, a parser provided determines identifiers ("portion identifiers") of at least some portions of XML data provided to an application, and makes the portion identifiers available to the application. As a result, the application may not need to construct the portion identifiers of various portions of the XML data of interest, and the programming complexity of applications can be reduced as a result.

(Para 32) In the case of event-based parsers, the portion identifiers can be provided along with the data elements as parameters of a single procedure call. In the case of object-based parsers, the portion identifiers can be made available in the data structures (random access memory) from which

applications typically access the data elements. In alternative embodiments, additional procedure calls (often referred to as function calls) can be used to provide the portion identifiers, as described below with examples.

(Para 33) A parser may construct the portion identifiers while parsing the XML data files. By constructing the portion identifiers while parsing the data files, the implementation of parsers also can be simplified.

(Para 34) Several aspects of the invention are described below with reference to examples for illustration. It should be understood that numerous specific details, relationships, and methods are set forth to provide a full understanding of the invention. One skilled in the relevant art, however, will readily recognize that the invention can be practiced without one or more of the specific details, or with other methods, etc. In other instances, well-known structures or operations are not shown in detail to avoid obscuring the invention.

(Para 35) 2. Example Environment

(Para 36) Figure 1 is a block diagram illustrating an example environment in which various aspects of the present invention can be implemented. Application 110 is shown interfacing with parser 130 for data elements stored in XML data file 140. Application 110 generally processes the data elements provided by parser 130. Parser 130 represents an example parser, which simplifies the implementation of application 110.

(Para 37) Various example embodiments described below can be implemented on workstations/servers/systems available from vendors such as Sun Microsystems, IBM, and Dell supporting the JAVA application environment. Java concepts are described in further detail in a book entitled, "JavaTM The Complete Reference, Fifth Edition" by Herbert Schildt, ISBN Number: 0-07-049543-2.

(Para 38) Application 110 generally needs to be implemented consistent with the interface (shown as 112) provided by parser 130. Parser 130 can be implemented by modifying any of the parsers (e.g., the event-based and object based parsers noted above in the background section) available in the marketplace to implement various aspects of the present invention. Alternatively, custom API (application programming interface) may be defined to suitably provide the XPath(s) associated with various data elements.

(Para 39) Some example interfaces (shown as path 112) between applications and parsers, and the manner in which the interfaces can be implemented is described below with examples. First, the manner in which parser 130 and application 110 operate according to several aspects of the present invention is described below first.

(Para 40) 3. Operation of Parser

(Para 41) Figure 2 is a flow chart illustrating the manner in which a parser may operate according to various aspects of the present invention. The flow chart is described with reference to Figure 1 merely for illustration. The flow chart begins in step 201 and control immediately passes to step 210.

(Para 42) In step 210, parser 130 may receive the identifier ("file identifier" to distinguish from the portion identifiers of portions data file) of an XML data file from an application. It should be appreciated that the file represents any data source (whether or not stored in secondary medium) and the file identifier identifies the data source.

(Para 43) In the example environment of Figure 1, parser 130 receives the file identifier of XML data file 140 from application 110 according to interface 112. Parser 130 may identify the XML data file 140 in the corresponding operating environment (e.g., Sun server with Java application environment), based on the received file identifier.

(Para 44) In step 220, parser 130 retrieves a data element from the XML data file. In general, the retrieval needs to be consistent with the medium and access mechanism by which the data in the XML data file can be accessed, and the retrieval may be performed in a known way.

(Para 45) In step 230, parser 130 determines XPath expression of the data element. In an embodiment, the loop of steps 220-250 is executed for each data element in the XML data file. The data elements are retrieved sequentially from the XML data file, and the XPath may be computed as each data element is retrieved. The content of the XML data file can be used in determining the XPath expression. The XPath expression can also be computed in a known way.

(Para 46) Continuing with Figure 2, in step 240, parser 130 may provide XPath expression associated with the data element to the application. The XPath expression may be provided using various interfaces (112). In an embodiment described below, a parser using an event-based parsing technique provides XPath to the application as a parameter while providing corresponding data element. In another embodiment below, parser 130 using object-based parsing technique provides XPath expression in the data structure representing XML data file 140 in RAM.

(Para 47) In yet another approach described below, parser 130 provides XPath of each data element using API procedure calls defined according to various aspects of the present invention. In the case of data elements having data values, the data value is contained in the provided XPath. Each of the approaches is described in further detail below.

(Para 48) In step 250, a determination may be made as to whether there are additional data elements to be provided to the application. Control passes to step 220 if such additional data elements are present, and to step 299 otherwise. The flow chart of Figure 2 ends in step 299. It may thus be appreciated that the flow chart of Figure 2 operates to provide XPath expressions associated

with atleast some data elements to applications. The description is continued with respect to the operation of application 110 in view of the availability of the XPath expressions.

(Para 49) 4. Operation of an Application

(Para 50) Figure 3 is a flow chart illustrating the manner in which an application may operate according to various aspects of the present invention. The flow chart is described with reference to Figure 1 merely for illustration. The flow chart begins in step 301 and control immediately passes to step 310.

(Para 51) In step 310, application 110 instructs a parser to parse an XML data file of interest. With reference to Figure 1, application 110 may specify an identifier of XML data file 140 while instructing parser 130. Any other initialization tasks, as necessary for the specific operating environment, may also be performed as a part of such instruction. In general, the instruction causes parser 130 to provide data elements contained in the specified data file to application 110.

(Para 52) In step 325, application 110 obtains XPath expression associated with an element from the parser. In embodiments described below, the XPath expression is obtained with respect to at least all the data elements, which have corresponding data values. However, alternative embodiments can be implemented in which XPath expressions are obtained in association with only some of the data elements of interest (e.g., by having the applications indicate such data elements of interest).

(Para 53) In step 330, application 110 may process the data element and XPath expression obtained from the parser. Such processing generally depends on the 'business logic' sought to be implemented by application 110. The XPath expression may be conveniently used to simplify the implementation of such business logic.

(Para 54) In step 340, application 110 determines if there are more elements to be obtained from XML data file. Control passes to step 325 if there are more elements to be obtained, and otherwise control passes to step 399, in which the flow chart ends. It may be appreciated that the implementation of applications can be simplified due to the availability of XPath. The description is continued with reference to an example illustrating the various XPaths provided to an application in the context of an example data file.

(Para 55) 5. XPaths for an Example XML Data File

(Para 56) Figure 4A depicts the contents of a XML data file containing 15 lines, numbered 401-415. The XPaths provided corresponding to the data elements in the 15 lines are depicted in the table of Figure 4B.

(Para 57) Figure 4B is shown containing a table having 2 columns 420 and 440. Column 420 is shown containing data elements for corresponding data in the XML data file of Figure 4A and column

440 is shown containing XPath of the data elements of column 420. For example, in row 421, XPath of data element ' Books' (under column 440) from the root is shown as /Books for the corresponding data shown in lines 402 and 415 of Figure 4A.

(Para 58) Lines 422-425 contain XPaths for the corresponding four data elements of lines 403-406 respectively. Similarly, lines 426-429 and 430-433 represent XPaths for the data elements of lines 407-410 and 411-414 respectively. An XPath /books/book would represent the data portion corresponding to all the three book elements.

(Para 59) It may be appreciated that the XPaths thus generated can be made available to applications using different interfaces, as described below. The description is continued with reference to an approach in which the interfaces provided by some prior parsers can be extended to provide the XPath expressions.

(Para 60) 6. Extensions to DOM Parsers

(Para 61) As noted above, DOM parser represents an object-based parser. DOM parser provides a pre-specified interface using which applications can obtain data elements from a data file of interest. DOM parser is described in a document entitled, "Effective XML" by Elliotte Rusty Harold, available from Addison-Wesley Professional, ISBN: 0321 504 06. The manner in which such a parser can be modified is illustrated below with reference to Figure 5A.

(Para 62) Figure 5A is shown containing Java code providing an outline for the implementation of an application interfacing with an object-based parser which provides portion identifiers (XPaths in this example) according to various aspects of the present invention. The pseudo-code is also used to illustrate the modifications may need to be performed to DOM parser to support various features of the present invention.

(Para 63) Lines 501, 505 and 507 are respectively shown importing the classes in a Java application code, available in `java.xml.parsers.*`, `org.w3c.dom.*`, and `java.util.Vector` respectively. The package `org.w3c.dom` contains functions for accessing (traversing, modifying, creating) a (data structure) DOM according to W3C standards.

(Para 64) Line 508 defines class DOM parsing as being public, and the corresponding body (of the class) is contained in lines 511-528, as shown. Line 509 defines a variable 'xpaths' as a vector. The xpaths variable is then used to store XPath, as described below.

(Para 65) Line 511 causes the execution of the code corresponding to lines 513-526 to be executed. Line 513 initiates a new instance of the DOM parser, and the corresponding handle is saved in variable `dbf`. In line 517, the factory instantiates the underlying registered DOM parser, and returns the pointer of the DOM parser class.

(Para 66) Line 520 specifies the file identifier of the XML data file to be parsed. Thus, the code of lines 513, 517 and 520 together perform the necessary initializations and completing instructing the DCM parser to parse a data file of interest.

(Para 67) In response, DCM parser provided/modified according to an aspect of the present invention parses the specified XML data file and load a data structure in the memory, with the data structure containing both the data elements and the corresponding XPath. A pointer to the data structure is returned to the application.

(Para 68) The data structure may be designed to store the corresponding XPath as well. Any convention can be used to store the XPath, and applications need to be designed consistently. An example convention is described below with reference to lines 522 and 524.

(Para 69) Line 522 calls procedure `traverse()`, and the corresponding code is provided in Figure 5B. As described below, `traverse()` procedure operates to fill the `xpaths` data structure with the XPath of each data element, as well the data element. Lines 523-525 merely print the XPath values of the data elements for illustration, however, more complex business logic can be employed to use the received XPath, as suitable for specific scenarios.

(Para 70) Figure 5B contains the code outline for `traverse()` in one embodiment. In line 555, the XPath corresponding to the data element is accessed by the procedure/ function called `node.getXPath()`, and stored in the vector `xpaths`. Accordingly, DCM parser may need to be implemented to construct the XPath while parsing the XML data file, and making the XPath value available by `getXPath()` call. The implementation of such procedure calls will be apparent to one skilled in the relevant arts by reading the disclosure provided herein.

(Para 71) In lines 550-590, the parser traverses through each node in the DCM data structure and computes corresponding XPath expressions. As may be appreciated, there are various types of nodes in a DCM tree. The `Node.DOCUMENT_NODE` is the hook node from which the entire DCM tree follows. `Node.ELEMENT_NODE` represents an element tag. For example, for the XML portion `<salutation>hello</salutation>`, a node of the type `ELEMENT_NODE` would be created in the DCM tree for the tag `salutation`, and a node of the type `TEXT_NODE` would be created for the value `hello`. Line 571 would be executed when the current node is of type `ELEMENT_NODE`. At line 572, all the immediate children of the current node is collected in `NodeList`. Lines 581 – 583 recursively call the method `traverse` for every node collected in the `NodeList`.

(Para 72) Thus, using techniques such as those described above, XPath can be provided to applications in the context of DCM-type parsers. The description is continued with an illustration of pseudo-code implementation of an interface between an application and an event-based push parser according to the present invention.

(Para 73) 7. Extensions to Event-based Push Parser

(Para 74) Figure 6 is shown containing Java Code for an application interfacing with a push based parser (e.g., SAX parser) according to an aspect of the present invention. As noted a parser in such an approach retrieves data in an XML data file without necessarily receiving a request for a next data element, and "pushes" the retrieved data element to the application (after the application has specified the identifier of the XML data file).

(Para 75) Lines 601, 603, 605 and 606 are respectively shown importing the classes available in source files `java.xml.parsers.*`, `org.xml.sax.*`, `org.xml.sax.helpers.*` and `java.util.Vector.*` respectively. Once imported, the classes in the source files can be referred to directly in the application code.

(Para 76) Line 607 defines class `SAXParsingXPath2` as being public, and the corresponding body (of the class) is contained in lines 609-645, as shown. Class `SAXParsingXPath2` represents an implementation of SAX Parser according to an aspect of the present invention which returns XPath corresponding to each data element as a parameter. It may be appreciated that the procedure may return XPath in addition to the attributes and values of the data element.

(Para 77) Line 615 initiates the execution of class `SAXParsingXPath2` corresponding to lines 609-645. Line 609 causes execution of the code corresponding to lines 611-620. Line 611 initiates a new instance of the SAX Parser-Factory, and the corresponding handle is saved in variable `spf`.

(Para 78) Line 613 initiates a new instance of the SAX parser and the corresponding handle is saved in variable `sp`. Line 615 creates a handle (saved in variable `handler`) while executing the code corresponding to the class `SAXParsingXPath2`.

(Para 79) Line 616 specifies the file identifier of the XML data file to be parsed. Since the application is assumed to be interfacing with a push parser, the parsing operations begin in response to execution of line 616, and the data elements of the specified data file (here `"../something.xml"`) are made available. Lines 617-620 are shown printing the XPath for each data element, even though more complex business logic can process the XPaths.

(Para 80) Lines 621-627, 629-635 and 637-643 represent classes which obtain the XPaths provided by SAX parser according to various aspects of the present invention, add the XPaths to the variable vector (as indicated by lines 625 and 633), and process the corresponding attributes and XPath. As may be readily observed, XPath corresponding to a data element is shown obtained as a parameter value in each of lines 621, 629 and 637.

(Para 81) Accordingly, the SAX parser may need to be implemented to construct the XPath while parsing the XML data file, and provide the XPath value as a parameter with each class. The description is continued with an illustration of pseudo-code implementation of an interface between an application and an event-based pull parser in an embodiment of the present invention.

(Para 82) 8. Extensions to Event-based Pull Parser

(Para 83) Figure 7 is shown containing Java Code for an application interfacing with a event based pull parser according to an aspect of the present invention. As noted, in such an approach an application requests the parser to retrieve each data element in an XML data file and the parser retrieves (e.g., from an XML document) the next data element (in sequential order) from the XML data file and provides the next data element to the application (after the application has specified the identifier of the XML data file).

(Para 84) Lines 701, 702, 703, and 704 are respectively shown importing the classes available in source files `java.io.*`, `javax.xml.stream.*`, `javax.xml.stream.events.*`, `java.util.Vector.*`. Once imported, the classes in the source files can be referred to directly in the application code.

(Para 85) Line 705 defines class `PullParsingXPath` as being public, and the corresponding body (of the class) is contained in lines 705-735, as shown. Class `PullParsingXPath` represents an implementation of event based PULL Parser according to an aspect of the present invention which returns XPath corresponding to each data element as a parameter.

(Para 86) Line 706 defines a variable ' `xpaths` ' as a vector. The `xpaths` variable is then used to store XPath, as described below.

(Para 87) Line 708, application request the parser to begin parsing by providing the file identifier. Lines 709-710 begins parsing and retrieves data elements from the XML data file and a variable `pullParser` is defined to contain the data elements.

(Para 88) The program loop in lines 711-725, the parser determines XPath for each data element and adds the XPath value to the vector `xpaths`. Lines 715-724 illustrate the manner in which different business logic can be applied (even though only a print statement is shown in all cases, for simplicity) for different node-types. Similarly, the for loop of lines 730-732 prints the `Xpaths` in the vector `xpaths`. In general, a programmer may provide a desired business logic instead of the print statements.

(Para 89) It may be appreciated that the embodiments described above with respect to Figures 5A and 6 represents extensions to conventional parsers, in which XPath values are obtained as parameters. However, various aspects of the present invention can be implemented using other approaches as well. For example, new parsers providing custom application programming interfaces (API) may be implemented, as described below with an example.

(Para 90) 9. Parsers With Custom API

(Para 91) Figures 8A-8F are shown containing tables illustrating the (names of) procedures (API) supported by an example parser in an embodiment of the present invention. Figures 8 and 9 respectively illustrate the manner in which the APIs of Figures 8A-8F can be used to implement

applications according to various aspects of the present invention. Each of the Figures 8A-8F, 8 and 9 is described in further detail below.

(Para 92) Lines 811-817 of Figure 8A indicate names of procedures accessible via XPathParserFactory API. The procedure of line 811 is used by an application to create an instance of the parser (811). The application can access procedures of push parser (line 812). The setFeature procedure of line 814 enables an application to assign a value to a property, as described below with reference to Figures 8 and 9 in further detail. The hasFeature procedure of line 813 checks whether a particular property is already set/initialized. The getFeature procedure of line 815 enables an application to retrieve a presently assigned value (and can be used after checking with hasFeature procedure).

(Para 93) The methods setProperty (line 816) and setFeature (line 814) in XPathParserFactory, set corresponding value for property and feature in the parser to the value passed during corresponding function calls. The methods are illustrated with examples below. However, the methods can be used with respect to other types of features and properties, as suited for the specific environment.

(Para 94) Lines 821-825 of Figure 8B correspond to names of procedures that can be used by an application if push based parser is instantiated using the procedure call of line 812. The procedures of lines 821-825 are described with reference to Figure 9 below.

(Para 95) Lines 841-844 of Figure 8C contain names of procedures which are implemented in push based parser in one embodiment. The startDocument procedure of 841 opens an XML data file specified as a parameter, and (the file identifier) is generally provided by an application. Line 844 is shown indicating the name of the procedure, which may close the XML document after parsing of data elements is completed.

(Para 96) One of lines 842 and 843 is executed by the parser to provide the XPath value associated with each data element. The emit procedure of line 842 is used if the application had previously indicated that XPaths for the attributes need not be grouped and provided (by using the setFeature procedure of line 814) while providing the XPath of the associated data element. In such a case, separate XPaths are provided for each attribute of the data element. The emit procedure of line 843 is used otherwise, in which case the attributes are returned as values associated with the data element (along with the XPath).

(Para 97) The NamespaceResolver class of Figure 8D is used to resolve the prefixes in the XPath expression to the namespace URIs. As may be appreciated, the elements in an XML document may or may not have namespaces, depending on how the XML document is created. The XPath expression for an element, which has a namespace should generally use a prefix, and the mapping of this prefix to the actual namespace of the node must be available in the NamespaceResolver, so that the XPath engine can use the NamespaceResolver to resolve the prefixes to namespace while evaluating the XPath expression.

(Para 98) For example, for the XML document:

(Para 99) `<?xml version="1.0" ?>`

(Para 100) `<sdlutation xmlns="foo">hello</sdlutation>`

(Para 101) the element – `sdlutation`, has a namespace URI = `foo`. The XPath expression for this element is `"/pfx:sdlutation"` and not `"/sdlutation"`, wherein the prefix `pfx` could be any name and not necessarily always be `pfx`. The expr `"/sdlutation"` is incorrect because, the element `sdlutation` in the XML document has a namespace URI, and `"/sdlutation"` would mean to look for the element named `sdlutation` which has no namespace.

(Para 102) Now, for the XPath engine to be able to evaluate the expr `"/pfx:sdlutation"` correctly, the prefix – `pfx` must have been bound to some namespace, and there must be a mechanism by which the XPath engine can resolve the prefix to a namespace URI. The `XPathPushParser`, while parsing the XML document, would bind the prefixes used in the XPath expr to the correct namespace URIs in the `NamespaceResolver`. This `NamespaceResolver` would then be made available to the XPath engine to evaluate the XPath expression correctly.

(Para 103) For the example noted above, when the `XPathPushParser` reports the xpath as `"/pfx:sdlutation"`, the application code can use the `NamespaceResolver` in the following way to resolve the namespace URI:

(Para 104) `String ns = resolver.resolveNamespacePrefix("pfx");`

(Para 105) The value of the variable `"ns"` would be `"foo"`.

(Para 106) When a feature `http://xpath-parser/features/group-attributes` is set to true, the XPath of all the attributes with their corresponding values, if any, on an element, would be grouped together as `XPathAttributes` (Figure 8E) and reported alongwith the XPath of the element using the method `emit(String xpath, String xpath Val, int eventType, XPathAttributes attr, NamespaceResolver nsResolver)`.

(Para 107) For every attribute found, the XPath expression of the attribute and the value of the attribute is used to create the function `XpathAttribute` of Figure 8F. The set of all such `XPathAttribute` functions can then be accessed from the function `XpathAttributes` using the function as follows:

(Para 108) `public void emit(String xpath, String xpath Val, int eventType,
XPathAttributes attr, NamespaceResolver nsResolver) {`

(Para 109) `if (attr != null) {`

(Para 110) `int len = attr.getLength();`

(Para 111) `for (int i=0; i<len; i++) {`

(Para 112) `XPathAttribute attribute = attr.item(i);`

(Para 113) `System.out.println("Attr XPath=" + attribute.getXPath());`

```

(Para 114)      System.out.println("Attr XPath="+attribute.getValue());
(Para 115)      }
(Para 116)      }
(Para 117)      }

```

(Para 118) The implementation of the procedures of Figures 8A-8F will be apparent to one skilled in the relevant arts by reading the disclosure provided herein. The description is continued with respect to the manner in which applications can be implemented using the custom API thus provided.

(Para 119) 10. Applications Using Parsers With Custom API

(Para 120) Figure 9 is shown containing Java code of an application using procedures related to event based parsing using the API described above with reference to Figures 8A-8C and 8E-8F. Lines 901-904 respectively import javax.xml.parsers.*, org.xml.sax.*, org.xml.sax.helpers.* and java.util.Vector libraries.

(Para 121) Line 905 defines class PushParsingXPath as being public, and the corresponding body (of the class) is contained in lines 907-933. Class PushParsingXPath represents an implementation of an event-based parser, which returns XPath corresponding to each data element according to an aspect of the present invention.

(Para 122) Line 906 defines a variable 'xpathes' as a vector. The xpathes variable is then used to store XPath, as described below.

(Para 123) Line 908 is shown initializing an instance of XPathParserFactory and the corresponding handle is stored in a variable xpf to enable access to procedure names indicated by lines 811-817 of Figure 8A.

(Para 124) Line 909 is shown initializing an instance of XPath based PUSH parser, and the corresponding handle is stored in a variable xpp, which enables the application to access corresponding procedures of 821-825.

(Para 125) Line 910 indicates to the parser that the XPaths are to be provided in a non-abbreviated format, consistent with the definition of the procedure call of line 814. As a result, each XPath is provided with the beginning of the XML data file as the root (e.g., as depicted in column 1020 of Figure 10A). On the other hand, if the value were set to true, the XPath are provided in abbreviated format, in which each XPath is defined with reference to a present node.

(Para 126) Continuing with reference to Figure 9, selfFeature procedure of line 911 requests the parser to provide XPath of attributes corresponding to a data element while parsing the XML data file by executing either of the emit procedures indicated by the lines 921-925 and 927-933.

(Para 127) Lines 912 and 913 initialize the application to process data elements provided by the parser (by the emit statement, noted above in line 842 and 843 of Figure 8C) and the corresponding handle is stored in variable xpdHandler.

(Para 128) Lines 914 and 915 together enable application to report any errors, which may occur during processing in the application. The corresponding handle is stored in a variable xpeHandler. In Line 917, application provides the file identifier for XML data file to the parser.

(Para 129) In line 919, the parser begins parsing of XML data file and procedures of Figures 8C, 8E and 8F are executed. In particular, as noted above, one of the emit statements of lines 842 and 843 is executed depending on the value of the attributes parameter, which is set to true in line 911. Thus, the statement of line 843 would be executed due to the set value.

(Para 130) The code of lines 921-925 and 927-931 respectively inherit the emit classes of lines 842 and 843. The xpaths.add() procedure is executed by each of the emit classes. However, more complex business logic can be employed, as will be apparent to one skilled in the relevant arts.

(Para 131) It may be observed that the above application is implemented with push parsers. Figure 10A illustrates the XPathS and values returned by the parser, as described briefly below.

(Para 132) Figure 10A is a table containing three columns node (data element) 1010, XPath 1020 and value 1030. The three columns illustrate the XPath and value returned for each data element when the attribute feature is set to true. Rows 1022-1033 correspond to each data element of the XML data file of Figure 4A. In particular, row 1024 indicates that the XML parsers of Figures 8A-8F would provide XPath of "/Books/Book(1)/@isbn" and a value of ' 123' for the attribute ' ISBN=123' shown in line 403 of Figure 4A. Similarly, as may be readily observed, lines 1028 and 1032 also contain data elements corresponding to attributes (of lines 407 and 411 of Figure 4A respectively).

(Para 133) Figure 10B is a table illustrating the operation of the parser when the attribute feature is set to false. Columns 1051-54 respectively represent node (i.e., data element), XPath, corresponding Value and XPathAttributes, as shown. Thus, the parser may provide the data represented by rows 1061-1071 to the application. As may be appreciated, in such a case, the parser does not report the XPath for the attributes in such a scenario.

(Para 134) In comparison to Figure 10A, it may be readily noted that XPathS are not provided for attributes in rows 1024, 1028 and 1032. However, the same attribute values are provided as parameters as indicated in rows 1063, 1066 and 1069 of Figure 10B.

(Para 135) It may be further noted that the XPathS of Figures 10A and 10B are shown in non-abbreviated format since the abbreviated feature is set to false in lines 810 and 907 respectively. However, by setting the abbreviated feature to true, the XPathS can be provided in the abbreviated format, and the corresponding XPathS are depicted in Figure 10C. The table of Figure 10D depicts the XPathS corresponding to a case in which abbreviated feature and attributes feature are both set to false.

(Para 136) The description is continued with reference to an embodiment in which the above features are implemented in the form of software instructions executing on a digital processing system.

(Para 137) 11. Software-driven Implementation

(Para 138) Figure 11 is a block diagram illustrating the details of how various aspects of the invention may be implemented substantially in the form of software in an embodiment of the present invention. System 1100 may contain one or more processors such as central processing unit (CPU) 1110, random access memory (RAM) 1120, secondary memory 1130, graphics controller 1160, display unit 1170, network interface 1180, and input interface 1190. All the components except display unit 1170 may communicate with each other over communication path 1150, which may contain several buses as is well known in the relevant arts. The components of Figure 11 are described below in further detail.

(Para 139) CPU 1110 may execute instructions stored in RAM 1120 to provide several features of the present invention. For example, the instructions may implement one or both of parsers and applications, described above. CPU 1110 may contain only a single general purpose-processing unit or several processing units. RAM 1120 may receive instructions from secondary memory 1130 using communication path 1150.

(Para 140) Graphics controller 1160 generates display signals (e.g., in RGB format) to display unit 1170 based on data/instructions received from CPU 1110. Display unit 1170 contains a display screen to display the images defined by the display signals. Input interface 1190 may correspond to a keyboard and/or mouse. Graphics controller 1160 and input interface 1190 may enable a user to interact directly with system 1100.

(Para 141) Secondary memory 1130 may contain hard drive 1135, flash memory 1136 and removable storage drive 1137. Secondary memory 1130 may store the data and software instructions, which enable system 1100 to provide several features in accordance with the present invention. Some or all of the data and instructions may be provided on removable storage unit 1140, and the data and instructions may be read and provided by removable storage drive 1137 to CPU

1110. Floppy drive, magnetic tape drive, CD-ROM drive, DVD Drive, Flash memory, removable memory chip (PCMCIA Card, EPROM) are examples of such removable storage drive 1137.

(Para 142) Removable storage unit 1140 may be implemented using medium and storage format compatible with removable storage drive 1137 such that removable storage drive 1137 can read the data and instructions. Thus, removable storage unit 1140 includes a computer readable storage medium having stored therein computer software and/or data.

(Para 143) In this document, the term "computer program product" is used to generally refer to removable storage unit 1140 or hard disk installed in hard drive 1135. These computer program products are means for providing software to system 1100. CPU 1110 may retrieve the software instructions, and execute the instructions to provide various features of the present invention as described above.

(Para 144) **12. Conclusion**

(Para 145) While various embodiments of the present invention have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of the present invention should not be limited by any of the above-described example embodiments, but should be defined only in accordance with the following claims and their equivalents.